CHAPTER 3

The MVC Pattern

In Chapter 7 we are going to start building a more complex ASP.NET MVC example. Before we start digging into the details of the ASP.NET MVC Framework, we want to make sure you are familiar with the MVC design pattern and the thinking behind it. In this chapter, we describe the following:

- The MVC architecture pattern
- Domain models and repositories
- Creating loosely coupled systems using dependency injection (DI)
- The basics of automated testing

You might already be familiar with some of the ideas and conventions we discuss in this chapter, especially if you have done advanced ASP.NET or C# development. If not, we encourage you to read this chapter carefully—a good understanding of what lies behind MVC can help put the features of the framework into context as we continue through the book.

The History of MVC

The term *model-view-controller* has been in use since the late 1970s and arose from the Smalltalk project at Xerox PARC where it was conceived as a way to organize some early GUI applications. Some of the fine detail of the original MVC pattern was tied to Smalltalk-specific concepts, such as *screens* and *tools*, but the broader concepts are still applicable to applications—and they are especially well suited to Web applications.

Interactions with an MVC application follow a natural cycle of user actions and view updates, where the view is assumed to be stateless. This fits nicely with the HTTP requests and responses that underpin a Web application.

Further, MVC forces a *separation of concerns*—the domain model and controller logic is decoupled from the user interface. In a Web application, this means that the mess of HTML is kept apart from the rest of the application, which makes maintenance and testing simpler and easier. It was Ruby on Rails that led to renewed mainstream interest in MVC and it remains the poster child for the MVC pattern. Many other MVC frameworks have since emerged and demonstrated the benefits of MVC—including, of course, ASP.NET MVC.

Understanding the MVC Pattern

In high-level terms, the MVC pattern means that an MVC application will be split into at least three pieces:

- *Models*, which contain or represent the data that users work with. These can be simple *view models*, which just represent data being transferred between views and controllers; or they can be *domain models*, which contain the data in a business domain as well as the operations, transformations, and rules for manipulating that data.
- *Views,* which are used to render some part of the model as a user interface.
- *Controllers*, which process incoming requests, perform operations on the model, and select views to render to the user.

Models are the definition of the universe your application works in. In a banking application, for example, the model represents everything in the bank that the application supports, such as accounts, the general ledger, and credit limits for customers—as well as the operations that can be used to manipulate the data in the model, such as depositing funds and making withdrawals from the accounts. The model is also responsible for preserving the overall state and consistency of the data—for example, making sure that all transactions are added to the ledger, and that a client doesn't withdraw more money than he is entitled to or more money than the bank has.

Models are also defined by what they are *not* responsible for: models don't deal with rendering UIs or processing requests—those are the responsibilities of *views* and *controllers*. *Views* contain the logic required to display elements of the model to the user—and nothing more. They have no direct awareness of the model and do not directly communicate with the model in any way. *Controllers* are the bridge between views and the model—requests come in from the client and are serviced by the controller, which selects an appropriate view to show the user and, if required, an appropriate operation to perform on the model.

Each piece of the MVC architecture is well-defined and self-contained—this is referred to as the *separation of concerns*. The logic that manipulates the data in the model is contained *only* in the model; the logic that displays data is *only* in the view, and the code that handles user requests and input is contained *only* in the controller. With a clear division between each of the pieces, your application will be easier to maintain and extend over its lifetime, no matter how large it becomes.

Understanding the Domain Model

The most important part of an MVC application is the domain model. We create the model by identifying the real-world entities, operations, and rules that exist in the industry or activity that our application must support, known as the *domain*.

We then create a software representation of the domain—the *domain model*. For our purposes, the domain model is a set of C# types (classes, structs, etc.), collectively known as the *domain types*. The operations from the domain are represented by the methods defined in the domain types, and the domain rules are expressed in the logic inside of these methods—or, as we saw in the previous chapter, by applying C# attributes to the methods. When we create an instance of a domain type to represent a specific piece of data, we create a *domain object*. Domain models are usually persistent and long-lived—there are lots of different ways of achieving this, but relational databases remain the most common choice.

In short, a domain model is the single, authoritative definition of the business data and processes within your application. A *persistent* domain model is also the authoritative definition of the state of your domain representation.

The domain model approach solves many of the problems that arise in the smart UI pattern. Our business logic is contained in one place—if you need to manipulate the data in your model or add a new process or rule, the domain model is the only part of your application that has to be changed.

Applying Domain-Driven Development

We have already described how a *domain model* represents the real world in your application, containing representations of your objects, processes, and rules. The domain model is the heart of an MVC application—everything else, including views and controllers, is just a means to interact with the domain model.

ASP.NET MVC does not dictate the technology used for the domain model. You are free to select any technology that will interoperate with the .NET Framework—and there are lots of choices. However, ASP.NET MVC does provide infrastructure and conventions to help connect the classes in the domain model with the controllers and views, and with the MVC Framework itself. There are three key features:

- *Model binding* is a convention-based feature that populates model objects automatically using incoming data, usually from an HTML form post.
- *Model metadata* lets you describe the meaning of your model classes to the framework. For example, you can provide human-readable descriptions of their properties or give hints about how they should be displayed. The MVC Framework can then automatically render a display or an editor UI for your model classes into your views.
- *Validation,* which is performed during model binding and applies rules that can be defined as metadata.

We briefly touched on model binding and validation when we built our first MVC application in Chapter 2—and we will return to these topics and investigate further in Chapters 22 and 23. For the moment, we are going to put the ASP.NET implementation of MVC aside and think about domain modeling as an activity in its own right. We are going to create a simple domain model using .NET and SQL Server, using a few core techniques from the world of *domain-driven development* (DDD).

Modeling an Example Domain

You have probably experienced the process of brainstorming a domain model. It usually involves developers, business experts, and copious quantities of coffee, cookies, and whiteboard pens. After a while, the people in the room converge on a common understanding and a first draft of the domain model emerges. (We are skipping over the many hours of disagreement and arguing that seems inevitable at this stage in the process. Suffice to say that the developers will spend the first hours askance at demands from the business experts for features that are taken directly from science fiction, while the business experts will express surprise and concern that time and cost estimates for the application are similar to what NASA requires to reach Mars. The coffee is essential in resolving such standoffs—eventually everyone's bladder is so full that progress will be made and compromises reached, just to bring the meeting to an end).

You might end up with something similar to Figure 3-5, which is the starting point for this example—a simple domain model for an auction application.

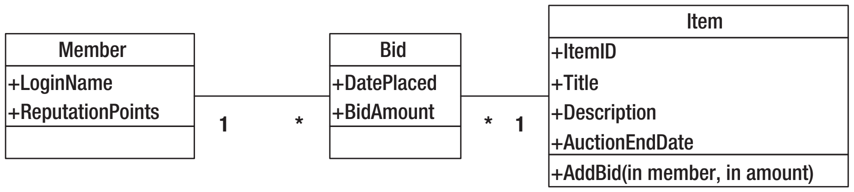

Figure 3-5. The first draft model for an auction application

This model contains a set of `Members`, which each hold a set of `Bids`. Each `Bid` is for an `Item`, and each `Item` can hold multiple `Bids` from different `Members`.

Ubiquitous Language

A key benefit of implementing your domain model as a distinct component is that you can adopt the language and terminology of your choice. You should try to find terminology for its objects, operations, and relationships that makes sense not just to developers, but to your business experts as well. We recommend that you adopt the domain terminology when it already exists—for example, if what a developer would refer to as *users* and *roles* are known as *agents* and *clearances* in the domain, we recommend you adopt the latter terms in your domain model.

And when modeling concepts that the domain experts do not have terms for, you should come to a common agreement about how you will refer to them, creating a ubiquitous language that runs throughout the domain model.

Developers tend to speak in the language of the code—the names of classes, database tables and so on. Business experts do not understand these terms—and nor should they have to. A business expert with a little technical knowledge is a dangerous thing, because he will be constantly filtering his requirements through his understanding of what the technology is capable of—when this happens, you do not get a true understanding of what the business requires.

This approach also helps to avoid overgeneralization in an application. Programmers have a tendency to want to model every possible business reality, rather than the specific one that the business requires. In the auction model, we thus might end up replacing *members* and *items* with a general notion of *resources* linked by *relationships*. When we create a domain model that is not constrained to match the domain being modeled, we miss the opportunity to gain any real insight in the business processes—and, in the future, we end up representing changes in business processes as awkward corner-cases in our elegant but overly abstract meta-world. Constraints are not limitations—they are insights that direct your development efforts in the right direction.

The link between the ubiquitous language and the domain model should not be a superficial one—DDD experts suggest that any change to the ubiquitous language should result in a change to the model. If you let the model drift out of sync with the business domain, you effectively create an intermediate language that maps from the model to the domain—and that spells disaster in the long term. You will create a special class of people who can speak both languages and they will then start filtering requirements through their incomplete understanding of both languages.

Aggregates and Simplification

Figure 3-5 provides a good starting point for our domain model, but it doesn't offer any useful guidance about implementing the model using C# and SQL Server. If we load a `Member` into memory, should we also load her `Bids` and the `Items` associated with them? And, if so, do we need to load all of the other `Bids` for those Items, and the `Members` who made those bids? When we delete an object, should we delete related objects, too—and, if so, which ones? If we choose to implement persistence using a document store instead of a relational database, which collections of objects would represent a single document? We don't know, and our domain model doesn't give us any of the answers to this question.

The DDD way of answering these questions is to arrange domain objects into groups called *aggregates*—Figure 3-6 shows how we might aggregate the objects in our auction domain model.

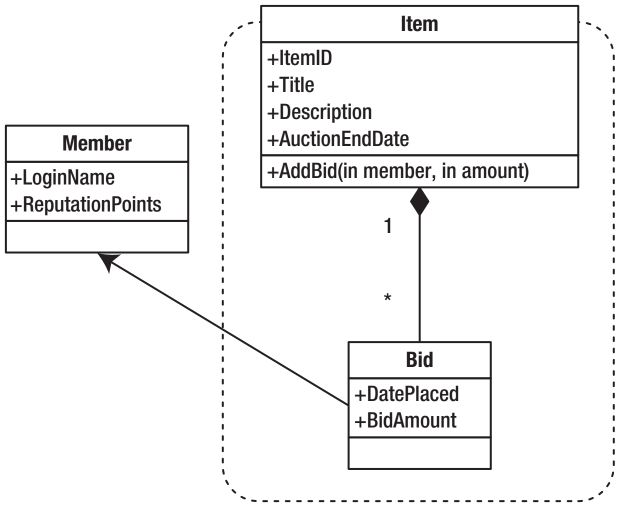

Figure 3-6. The auction domain model with aggregates

An aggregate entity groups together several domain model objects—there is a *root entity* that is used to identify the entire aggregate, and it acts as the "boss" for validation and persistence operations. The aggregate is treated as a single unit with regard to data changes, so we need to create aggregates that represent relationships that make sense in the context of the domain model and create operations that correspond logically to real business processes—that is, we need to create aggregates by grouping objects that are changed as a group.

A key DDD rule is that objects outside of a particular instance of an aggregate can hold persistent references to only the root entity, not to any other object inside of the aggregate (in fact, the identity of a non-root object need be unique only within its aggregate). This rule reinforces the notion of treating the objects inside an aggregate as a single unit.

In our example, `Members` and `Items` are both aggregate roots, whereas `Bids` can be accessed only in the context of the `Item` that is the root entity of their aggregate. `Bids` are allowed to hold references to `Members` (which are root entities), but `Members` can't directly reference `Bids` (because they are not).

One of the benefits of aggregates is that it simplifies the set of relationships between objects in the domain model—and often, this can give additional insight into the nature of the domain that is being modeled. In essence, creating aggregates constrains the relationships between domain model objects so that they are more like the relationships that exist in the real-world domain. Listing 3-1 illustrates how our domain model might look like when expressed in C#.

Listing 3-1. The C# Auction Domain Model

```
public class Member {
    public string LoginName { get; set; } // The unique key
    public int ReputationPoints { get; set; }
}

public class Item {
    public int ItemID { get; private set; } // The unique key
    public string Title { get; set; }
    public string Description { get; set; }
    public DateTime AuctionEndDate { get; set; }
    public IList<Bid> Bids { get; set; }
}

public class Bid {
    public Member Member { get; set; }
    public DateTime DatePlaced { get; set; }
    public decimal BidAmount { get; set; }
}
```

Notice how we are easily able to capture the unidirectional nature of the relationship between `Bids` and `Members`. We have also been able to model some other constraints—for example, `Bids` are immutable (representing the common auction convention that bids can't be changed once they are made). Applying aggregation has allowed us to create a more useful and accurate domain model, which we have been able to represent in C# with ease.

In general, aggregates add structure and accuracy to a domain model. They make it easier to apply validation (the root entity becomes responsible for validating the state of all objects in the aggregate) and are obvious units for persistence. And, because aggregates are essentially the atomic units of our domain model, they are also suitable units for transaction management and cascade deletes from databases.

On the other hand, they impose restrictions that can sometimes appear artificial—because often they *are* artificial. Aggregates arise naturally in document databases, but they are not a native concept in SQL Server, nor in most ORM tools, so to implement them well, your team will need discipline and effective communication.

Defining Repositories

At some point, we will need to add persistence for our domain model—this will usually be done through a relational, object, or document database. Persistence is not part of our domain model—it is an *independent* or *orthogonal concern* in our separation of concerns pattern. This means that we don't want to mix the code that handles persistence with the code that defines the domain model.

The usual way to enforce separation between the domain model and the persistence system is to define *repositories*—these are object representations of the underlying database (or file store or whatever you have chosen). Rather than work directly with the database, the domain model calls the methods defined by the repository, which in turn makes calls to the database to store and retrieve the model data—this allows us to isolate the model from the implementation of the persistence.

The convention is to define separate data models for each aggregate, because aggregates are the natural unit for persistence. In the case of our auction, for example, we might create two repositories—one for `Members` and one for `Items` (note that we don't need a repository for `Bids`, because they will be persisted as part of the `Items` aggregate). Listing 3-2 shows how these repositories might be defined.

Listing 3-2. C# Repository Classes for the Member and Item Domain Classes

```
public class MembersRepository {

    public void AddMember(Member member) {
        /* Implement me */
    }
    public Member FetchByLoginName(string loginName) {
        /* Implement me */ return null;
    }
    public void SubmitChanges() {
        /* Implement me */
    }
}

public class ItemsRepository {

    public void AddItem(Item item) {
        /* Implement me */
    }

    public Item FetchByID(int itemID) {
        /* Implement me */  return null;
    }

    public IList<Item> ListItems(int pageSize, int pageIndex) {
        /* Implement me */ return null;
    }
    public void SubmitChanges() {
        /* Implement me */
    }
}
```

Notice that the repositories are concerned only with loading and saving data—they contain no domain logic at all. We can complete the repository classes by adding statements to each method that perform the store and retrieve operations for the appropriate persistence mechanism. In Chapter 7, we will start to build a more complex and realistic MVC application and, as part of that process, we'll show you how to use the Entity Framework to implement your repositories.

Building Loosely Coupled Components

As we have said, one of most important features of the MVC pattern is that it enables separation of concerns. We want the components in our application to be as independent as possible and to have as few interdependencies as we can manage.

In our ideal situation, each component knows nothing about any other component and only deals with other areas of the application through abstract interfaces. This is known as *loose coupling*, and it makes testing and modifying our application easier.

A simple example will help put things in context. If we are writing a component called `MyEmailSender` that will send e-mails, we would implement an interface that defines all of the public functions required to send an e-mail, which we would call `IEmailSender`.

Any other component of our application that needs to send an e-mail—let's say a password reset helper called `PasswordResetHelper`, can then send an e-mail by referring only to the methods in the interface. There is no direct dependency between `PasswordResetHelper` and `MyEmailSender`, as shown by Figure 3-7.

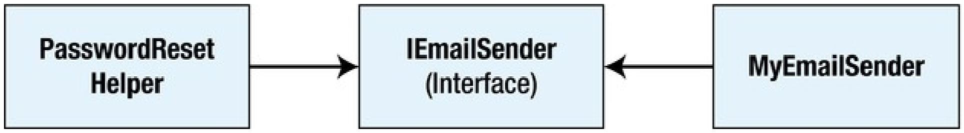

Figure 3-7. Using interfaces to decouple components

By introducing `IEmailSender`, we ensure that there is no direct dependency between `PasswordResetHelp` and `MyEmailSender`. We could replace `MyEmailSender` with another e-mail provider or even use a mock implementation for testing purposes. We return to the topic of mock implementations in Chapter 6.

■ **Note** Not every relationship needs to be decoupled using an interface. The decision is really about how complex the application is, what kind of testing is required, and what the long-term maintenance is likely to be. For example, we might choose not to decouple the controllers from the domain model in a small and simple ASP.NET MVC application.

Using Dependency Injection

Interfaces help us decouple components, but we still face a problem—C# doesn't provide a built-in way to easily create objects that implement interfaces, except to create an instance of the concrete component. We end up with the code in Listing 3-3.

Listing 3-3. Instantiating Concrete Classes to Get an Interface Implementation

```
public class PasswordResetHelper {

    public void ResetPassword() {

        IEmailSender mySender = new MyEmailSender();

        //...call interface methods to configure e-mail details...

        mySender.SendEmail();
    }
}
```

We are only part of the way to loosely coupled components—the `PasswordResetHelper` class is configuring and sending e-mails through the `IEmailSender` interface, but to create an object that implements that interface, it had to create an instance of `MyEmailSender`.

We have made things worse—now `PasswordResetHelper` depends on `IEmailSender` and `MyEmailSender`, as shown in Figure 3-8.

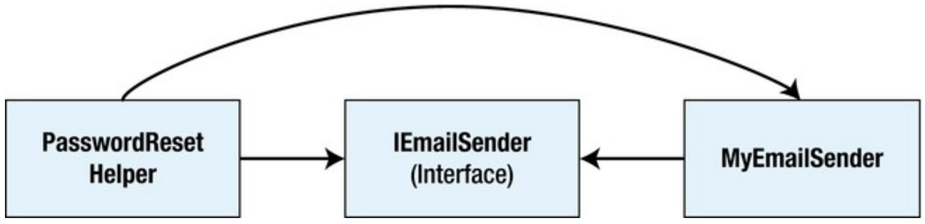

Figure 3-8. Components which are tightly coupled after all

What we need is a way to obtain objects that implement a given interface without having to create the implementing object directly. The solution to this problem is called *dependency injection* (DI), also known as *Inversion of Control* (IoC).